09 | 瞧一瞧Linux：Linux的自旋锁和信号量如何实现？

2021-05-28 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 16:58 大小 15.55M

你好，我是 LMOS。

上节课，我们学习了解决数据同步问题的思路与方法。Linux 作为成熟的操作系统内核，当然也有很多数据同步的机制，它也有原子变量、开启和关闭中断、自旋锁、信号量。

那今天我们就来探讨一下这些机制在 Linux 中的实现。看看 Linux 的实现和前面我们自己的实现有什么区别，以及 Linux 为什么要这么实现，这么实现背后的机理是什么。

Linux 的原子变量

首先，我们一起来看看 Linux 下的原子变量的实现，在 Linux 中，有许多共享的资源可能只是一个简单的整型数值。

例如在文件描述符中，需要包含一个简单的计数器。这个计数器表示有多少个应用程序打开了文件。在文件系统的 open 函数中，将这个计数器变量加 1；在 close 函数中，将这个计数器变量减 1。

如果单个进程执行打开和关闭操作，那么这个计数器变量不会出现问题，但是 Linux 是支持多进程的系统，如果有多个进程同时打开或者关闭文件，那么就可能导致这个计数器变量多加或者少加，出现错误。

为了避免这个问题，Linux 提供了**一个原子类型变量 atomic_t**。该变量的定义如下。

复制代码

```
typedef struct {
    int counter;
} atomic_t;//常用的32位的原子变量类型
#ifdef CONFIG_64BIT
typedef struct {
    s64 counter;
} atomic64_t;//64位的原子变量类型
#endif
```

上述代码自然不能用普通的代码去读写加减，而是要用 Linux 专门提供的接口函数去操作，否则就不能保证原子性了，代码如下。

复制代码

```
//原子读取变量中的值
static __always_inline int arch_atomic_read(const atomic_t *v)
{
    return __READ_ONCE((v)->counter);
}
//原子写入一个具体的值
static __always_inline void arch_atomic_set(atomic_t *v, int i)
{
    __WRITE_ONCE(v->counter, i);
}
//原子加上一个具体的值
static __always_inline void arch_atomic_add(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "addl %1,%0"
                 : "+m" (v->counter)
                 : "ir" (i) : "memory");
}
//原子减去一个具体的值
```

```
static __always_inline void arch_atomic_sub(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "subl %1,%0"
                 : "+m" (v->counter)
                 : "ir" (i) : "memory");
}
//原子加1
static __always_inline void arch_atomic_inc(atomic_t *v)
{
    asm volatile(LOCK_PREFIX "incl %0"
                 : "+m" (v->counter) :: "memory");
}
//原子减1
static __always_inline void arch_atomic_dec(atomic_t *v)
{
    asm volatile(LOCK_PREFIX "decl %0"
                 : "+m" (v->counter) :: "memory");
}
```

Linux 原子类型变量的操作函数有很多，这里我只是介绍了最基础的几个函数，其它的原子类型变量操作也依赖于上述几个基础的函数。

你会发现，Linux 的实现也同样采用了 x86 CPU 的原子指令，LOCK_PREFIX 是一个宏，根据需要展开成“lock;”或者空串。**单核心 CPU 是不需要 lock 前缀的，只要在多核心 CPU 下才需要加上 lock 前缀。**

剩下 __READ_ONCE，__WRITE_ONCE 两个宏，我们来看看它们分别做了什么，如下所示。

复制代码

```
#define __READ_ONCE(x)  \
(*(const volatile __unqual_scalar_typeof(x) *)&(x))
#define __WRITE_ONCE(x, val) \
do {*(volatile typeof(x) *)&(x) = (val);} while (0)
//__unqual_scalar_typeof表示声明一个非限定的标量类型，非标量类型保持不变。说人话就是返回›
//如果 x 是int类型则返回“int”
#define __READ_ONCE(x)  \
(*(const volatile int *)&(x))
#define __WRITE_ONCE(x, val) \
do {*(volatile int *)&(x) = (val);} while (0)
```

结合刚才的代码，我给你做个解读。Linux 定义了 __READ_ONCE，__WRITE_ONCE 这两个宏，是对代码封装并利用 GCC 的特性对代码进行检查，把让错误显现在编译阶段。其中的“volatile int *”是为了提醒编译器：**这是对内存地址读写，不要有优化动作，每次都必须强制写入内存或从内存读取。**

Linux 控制中断

Linux 中有很多场景，需要在关中断下才可以安全执行一些操作。

比如，多个中断处理程序需要访问一些共享数据，一个中断程序在访问数据时必须保证自身（中断嵌套）和其它中断处理程序互斥，否则就会出错。再比如，设备驱动程序在设置设备寄存器时，也必须让 CPU 停止响应中断。

Linux 控制 CPU 响应中断的函数如下。

复制代码

```
//实际保存eflags寄存器
extern __always_inline unsigned long native_save_fl(void){
    unsigned long flags;
    asm volatile("# __raw_save_flags\n\t"
                 "pushf ; pop %0":"=rm"(flags)::"memory");
    return flags;
}
//实际恢复eflags寄存器
extern inline void native_restore_fl(unsigned long flags){
    asm volatile("push %0 ; popf"::"g"(flags):"memory","cc");
}
//实际关中断
static __always_inline void native_irq_disable(void){
    asm volatile("cli":::"memory");
}
//实际开启中断
static __always_inline void native_irq_enable(void){
    asm volatile("sti":::"memory");
}
//arch层关中断
static __always_inline void arch_local_irq_disable(void){
    native_irq_disable();
}
//arch层开启中断
static __always_inline void arch_local_irq_enable(void){
    native_irq_enable();
}
//arch层保存eflags寄存器
```

```
static __always_inline unsigned long           arch_local_save_flags(void){
    return native_save_fl();
}
//arch层恢复eflags寄存器
static  __always_inline void arch_local_irq_restore(unsigned long flags){
    native_restore_fl(flags);
}
//实际保存eflags寄存器并关中断
static __always_inline unsigned long arch_local_irq_save(void){
    unsigned long flags = arch_local_save_flags();
    arch_local_irq_disable();
    return flags;
}
//raw层关闭开启中断宏
#define raw_local_irq_disable()     arch_local_irq_disable()
#define raw_local_irq_enable()      arch_local_irq_enable()
//raw层保存恢复eflags寄存器宏
#define raw_local_irq_save(flags)           \
    do {                        \
        typecheck(unsigned long, flags);    \
        flags = arch_local_irq_save();      \
    } while (0)

#define raw_local_irq_restore(flags)            \
    do {                        \
        typecheck(unsigned long, flags);    \
        arch_local_irq_restore(flags);      \
    } while (0)

#define raw_local_save_flags(flags)         \
    do {                        \
        typecheck(unsigned long, flags);    \
        flags = arch_local_save_flags();    \
    } while (0)
//通用层接口宏
#define local_irq_enable()              \
    do { \
        raw_local_irq_enable();         \
    } while (0)

#define local_irq_disable()             \
    do {                        \
        raw_local_irq_disable();        \
    } while (0)

#define local_irq_save(flags)               \
    do {                        \
        raw_local_irq_save(flags);      \
    } while (0)

#define local_irq_restore(flags)            \
    do {                        \
```

```
81          raw_local_irq_restore(flags);       \
82
```

可以发现，Linux 中通过定义的方式对一些底层函数进行了一些包装，为了让你抓住重点，前面这些宏我去掉了和中断控制无关的额外操作，详细信息你可以参阅相关代码。

编译 Linux 代码时，编译器自动对宏进行展开。其中，do{}while(0)是 Linux 代码中一种常用的技巧，do{}while(0) 表达式会保证{}中的代码片段执行一次，保证宏展开时这个代码片段是一个整体。

带 native_ 前缀之类的函数则跟我们之前实现的 hal_ 前缀对应，而 Linux 为了支持不同的硬件平台，做了多层封装。

Linux 自旋锁

Linux 也是支持多核心 CPU 的操作系统内核，因此 Linux 也需要自旋锁来对系统中的共享资源进行保护。同一时刻，只有获取了锁的进程才能使用共享资源。

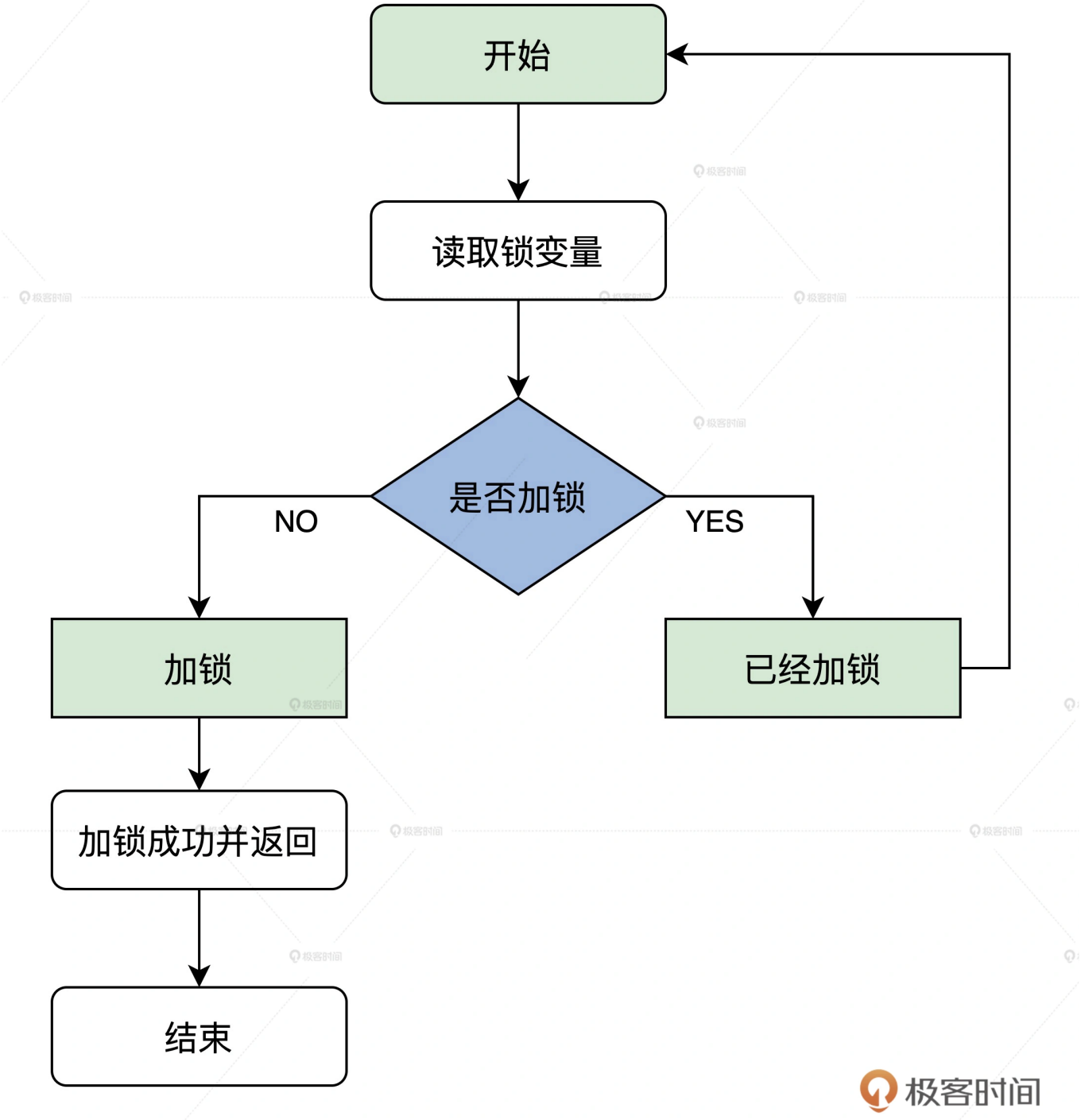
根据上节课对自旋锁算法的理解，自旋锁不会引起加锁进程睡眠，如果自旋锁已经被别的进程持有，加锁进程就需要一直循环在那里，查看是否该自旋锁的持有者已经释放了锁，"自旋"一词就是因此而得名。

Linux 有多种自旋锁，我们这里只介绍两种，**原始自旋锁和排队自旋锁**，它们底层原理和我们之前实现的没什么不同，但多了一些优化和改进，下面我们一起去看看。

Linux 原始自旋锁

我们先看看 Linux 原始的自旋锁，Linux 的原始自旋锁本质上用一个整数来表示，值为 1 代表锁未被占用，为 0 或者负数则表示被占用。

你可以结合上节课的这张图，理解后面的内容。当某个 CPU 核心执行进程请求加锁时，如果锁是未加锁状态，则加锁，然后操作共享资源，最后释放锁；如果锁已被加锁，则进程并不会转入睡眠状态，而是循环等待该锁，一旦锁被释放，则第一个感知此信息的进程将获得锁。



自旋锁原理示意图

我们先来看看 Linux 原始自旋锁的数据结构，为方便你阅读，我删除了用于调试的数据字段，代码如下。

复制代码

```
//最底层的自旋锁数据结构
typedef struct{
volatile unsigned long lock;//真正的锁值变量，用volatile标识
}spinlock_t;
```

Linux 原始自旋锁数据结构封装了一个 unsigned long 类型的变量。有了数据结构，我们再来看看操作这个数据结构的函数，即自旋锁接口，代码如下。

复制代码

```
#define spin_unlock_string \
    "movb $1,%0" \ //写入1表示解锁
    :"=m" (lock->lock) : : "memory"

#define spin_lock_string \
  "\n1:\t" \
    "lock ; decb %0\n\t" \ //原子减1
  "js 2f\n" \    //当结果小于0则跳转到标号2处，表示加锁失败
    ".section .text.lock,\"ax\"\n" \ //重新定义一个代码段，这是优化技术，避免后面的代码
  "2:\t" \
    "cmpb $0,%0\n\t" \  //和0比较
    "rep;nop\n\t" \  //空指令
    "jle 2b\n\t" \   //小于或等于0跳转到标号2
    "jmp 1b\n" \   //跳转到标号1
    ".previous"
//获取自旋锁
static inline void spin_lock(spinlock_t*lock){
    __asm__ __volatile__(
    spin_lock_string
    :"=m"(lock->lock)::"memory"
    );
}
//释放自旋锁
static inline void spin_unlock(spinlock_t*lock){
__asm__ __volatile__(
    spin_unlock_string
    );
}
```

上述代码中用 spin_lock_string、spin_unlock_string 两个宏，定义了获取、释放自旋锁的汇编指令。spin_unlock_string 只是简单将锁值变量设置成 1，表示释放自旋锁，spin_lock_string 中并没有像我们 Cosmos 一样使用 xchg 指令，而是使用了 decb 指令，这条指令也能原子地执行减 1 操作。

开始锁值变量为 1 时，执行 decb 指令就变成了 0，0 就表示加锁成功。如果小于 0，则表示有其它进程已经加锁了，就会导致循环比较。

Linux 排队自旋锁

现在我们再来看看 100 个进程获取同一个自旋锁的情况，开始 1 个进程获取了自旋锁 L，后面继续来了 99 个进程，它们都要获取自旋锁 L，但是它们必须等待，这时第 1 进程释放了自旋锁 L。请问，这 99 个进程中谁能先获取自旋锁 L 呢？

答案是不确定，因为这个次序依赖于哪个 CPU 核心能最先访问内存，而哪个 CPU 核心可以访问内存是由**总线仲裁协议**决定的。

很有可能最后来的进程最先获取自旋锁 L，这对其它等待的进程极其不公平，为了解决获取自旋锁的**公平性**，Linux 开发出了排队自旋锁。

你可以这样理解，想要给进程排好队，就需要确定顺序，也就是进程申请获取锁的先后次序，Linux 的排队自旋锁通过保存这个信息，就能更公平地调度进程了。

为了保存顺序信息，排队自旋锁重新定义了数据结构。

复制代码

```
1  //RAW层的自旋锁数据结构
2  typedef struct raw_spinlock{
3      unsigned int slock;//真正的锁值变量
4  }raw_spinlock_t;
5  //最上层的自旋锁数据结构
6  typedef struct spinlock{
7      struct raw_spinlock rlock;
8  }spinlock_t;
9  //Linux没有这样的结构，这只是为了描述方便
10 typedef struct raw_spinlock{
11     union {
12         unsigned int slock;//真正的锁值变量
13         u16 owner;
14         u16 next;
15     }
16 }raw_spinlock_t;
```

slock 域被分成两部分，分别保存**锁持有者**和**未来锁申请者**的序号，如上述代码 10～16 行所示。

只有 next 域与 owner 域相等时，才表示自旋锁处于未使用的状态（此时也没有进程申请该锁）。在排队自旋锁初始化时，slock 被置为 0，即 next 和 owner 被置为 0，Linux 进程执行申请自旋锁时，原子地将 next 域加 1，并将原值返回作为自己的序号。

如果返回的序号等于申请时的 owner 值，说明自旋锁处于未使用的状态，则进程直接获得锁；否则，该进程循环检查 owner 域是否等于自己持有的序号，一旦相等，则表明锁轮到自己获取。

进程释放自旋锁时，原子地将 owner 域加 1 即可，下一个进程将会发现这一变化，从循环状态中退出。进程将严格地按照申请顺序依次获取排队自旋锁。这样一来，原先进程无序竞争的乱象就迎刃而解了。

复制代码

```
static inline void __raw_spin_lock(raw_spinlock_t*lock){
int inc = 0x00010000;
int tmp;
__asm__ __volatile__(
"lock ; xaddl %0, %1\n" //将inc和slock交换，然后 inc=inc+slock
                        //相当于原子读取next和owner并对next+1
"movzwl %w0, %2\n\t"//将inc的低16位做0扩展后送tmp tmp=(u16)inc
"shrl $16, %0\n\t" //将inc右移16位 inc=inc>>16
"1:\t"
"cmpl %0, %2\n\t" //比较inc和tmp，即比较next和owner
"je 2f\n\t" //相等则跳转到标号2处返回
"rep ; nop\n\t" //空指令
"movzwl %1, %2\n\t" //将slock的低16位做0扩展后送tmp 即tmp=owner
"jmp 1b\n" //跳转到标号1处继续比较
"2:"
:"+Q"(inc),"+m"(lock->slock),"=r"(tmp)
::"memory","cc"
);
}
#define UNLOCK_LOCK_PREFIX LOCK_PREFIX
static inline void __raw_spin_unlock(raw_spinlock_t*lock){
__asm__ __volatile__(
UNLOCK_LOCK_PREFIX"incw %0"//将slock的低16位加1 即owner+1
:"+m"(lock->slock)
::"memory","cc");
}
```

上述代码中的注释已经描述得很清楚了，每条指令都有注解，供你参考。这里需要注意的是 Linux 为了避免差异性，在 spinlock_t 结构体中包含了 raw_spinlock_t，而在 raw_spinlock_t 结构体中并没使用 next 和 owner 字段，而是在代码中直接操作 slock 的高 16 位和低 16 位来实现的。

不知道你有没有过这样的经历？当你去银行办事，又发现人很多时，你很可能会选择先去处理一些别的事情，等过一会人比较少了，再来办理我们自己的业务。

其实，在使用自旋锁时也有同样的情况，当一个进程发现另一个进程已经拥有自己所请求的自旋锁时，就自愿放弃，转而做其它别的工作，并不想在这里循环等待，浪费自己的时间。

对于这种情况，Linux 同样提供了相应的自旋锁接口，如下所示。

复制代码

```
static inline int __raw_spin_trylock(raw_spinlock_t*lock){
    int tmp;
    int new;
    asm volatile(
    "movl %2,%0\n\t"//tmp=slock
    "movl %0,%1\n\t"//new=tmp
    "roll $16, %0\n\t"//tmp循环左移16位，即next和owner交换了
    "cmpl %0,%1\n\t"//比较tmp和new即（owner、next）?=（next、owner）
    "jne 1f\n\t" //不等则跳转到标号1处
    "addl $0x00010000, %1\n\t"//相当于next+1
    "lock ; cmpxchgl %1,%2\n\t"//new和slock交换比较
    "1:"
    "sete %b1\n\t" //new = eflags.ZF位，ZF取决于前面的判断是否相等
    "movzbl %b1,%0\n\t" //tmp = new
    :"=&a"(tmp),"=Q"(new),"+m"(lock->slock)
    ::"memory","cc");
    return tmp;
}
int __lockfunc _spin_trylock(spinlock_t*lock){
    preempt_disable();
    if(_raw_spin_trylock(lock)){
        spin_acquire(&lock->dep_map,0,1,_RET_IP_);
        return 1;
    }
    preempt_enable();
    return 0;
}
#define spin_trylock(lock) __cond_lock(lock, _spin_trylock(lock))
```

_cond_lock 只用代码静态检查工作，一定要明白 _spin_trylock 返回 1 表示尝试加锁成功，可以安全的地问共享资源了；返回值为 0 则表示尝试加锁失败，不能操作共享资源，应该等一段时间，再次尝试加锁。

Linux 信号量

Linux 中的信号量同样是用来保护共享资源，能保证资源在一个时刻只有一个进程使用，这是单值信号量。也可以作为资源计数器，比如一种资源有五份，同时最多可以有五个进程，这是多值信号量。

单值信号量，类比于私人空间一次只进去一个人，其信号量的值初始值为 1，而多值信号量，相当于是客厅，可同时容纳多个人。其信号量的值初始值为 5，就可容纳 5 个人。

信号量的值为正的时候。所申请的进程可以锁定使用它。若为 0，说明它被其它进程占用，申请的进程要进入睡眠队列中，等待被唤醒。所以信号量最大的优势是既**可以使申请失败的进程睡眠，还可以作为资源计数器使用。**

我们先来看看 Linux 实现信号量所使用的数据结构，如下所示：

复制代码

```
struct semaphore{
    raw_spinlock_t lock;//保护信号量自身的自旋锁
    unsigned int count;//信号量值
    struct list_head wait_list;//挂载睡眠等待进程的链表
};
```

下面我们就跟着 Linux 信号量接口函数，一步步探索 Linux 信号量工作原理，和它对进程状态的影响，先来看看 Linux 信号量的使用案例，如下所示。

复制代码

```
#define down_console_sem() do { \
    down(&console_sem);\
} while (0)
static void __up_console_sem(unsigned long ip) {
    up(&console_sem);
}
#define up_console_sem() __up_console_sem(_RET_IP_)
//加锁console
void console_lock(void)
{
    might_sleep();
    down_console_sem();//获取信号量console_sem
    if (console_suspended)
        return;
    console_locked = 1;
```

```
    console_may_schedule = 1;
}
//解锁console
void console_unlock(void)
{
    static char ext_text[CONSOLE_EXT_LOG_MAX];
    static char text[LOG_LINE_MAX + PREFIX_MAX];
    //……删除了很多代码
    up_console_sem();//释放信号量console_sem
    raw_spin_lock(&logbuf_lock);
    //……删除了很多代码
}
```

为了简单说明问题，我删除了很多代码，上面代码中以 console 驱动为例说明了信号量的使用。

在 Linux 源代码的 kernel/printk.c 中，使用宏 DEFINE_SEMAPHORE 声明了一个单值信号量 console_sem，也可以说是**互斥锁**，它用于保护 console 驱动列表 console_drivers 以及同步对整个 console 驱动的访问。

其中定义了宏 down_console_sem() 来获得信号量 console_sem，定义了宏 up_console_sem() 来释放信号量 console_sem，console_lock 和 console_unlock 函数是用于互斥访问 console 驱动的，核心操作就是调用前面定义两个宏。

上面的情景中，down_console_sem() 和 up_console_sem() 宏的核心主要是调用了信号量的接口函数 down、up 函数，完成获取、释放信号量的核心操作，代码如下。

复制代码

```
static inline int __sched __down_common(struct semaphore *sem, long state,long
{
    struct semaphore_waiter waiter;
    //把waiter加入sem->wait_list的头部
    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = current;//current表示当前进程，即调用该函数的进程
    waiter.up = false;
    for (;;) {
        if (signal_pending_state(state, current))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_current_state(state);//设置当前进程的状态，进程睡眠，即先前__down函数中
        raw_spin_unlock_irq(&sem->lock);//释放在down函数中加的锁

```

```
16          timeout = schedule_timeout(timeout);//真正进入睡眠
17          raw_spin_lock_irq(&sem->lock);//进程下次运行会回到这里，所以要加锁
18          if (waiter.up)
19              return 0;
20      }
21   timed_out:
22      list_del(&waiter.list);
23      return -ETIME;
24   interrupted:
25      list_del(&waiter.list);
26      return -EINTR;
27
28      //为了简单起见处理进程信号（signal）和超时的逻辑代码我已经删除
29  }
30  //进入睡眠等待
31  static noinline void __sched __down(struct semaphore *sem)
32  {
33      __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
34  }
35  //获取信号量
36  void down(struct semaphore *sem)
37  {
38      unsigned long flags;
39      //对信号量本身加锁并关中断，必须另一段代码也在操作该信号量
40      raw_spin_lock_irqsave(&sem->lock, flags);
41      if (likely(sem->count > 0))
42          sem->count--;//如果信号量值大于0,则对其减1
43      else
44          __down(sem);//否则让当前进程进入睡眠
45      raw_spin_unlock_irqrestore(&sem->lock, flags);
46  }
47  //实际唤醒进程
48  static noinline void __sched __up(struct semaphore *sem)
49  {
50      struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list, struct
51      //获取信号量等待链表中的第一个数据结构semaphore_waiter，它里面保存着睡眠进程的指针
52      list_del(&waiter->list);
53      waiter->up = true;
54      wake_up_process(waiter->task);//唤醒进程重新加入调度队列
55  }
56  //释放信号量
57  void up(struct semaphore *sem)
58  {
59      unsigned long flags;
60      //对信号量本身加锁并关中断，必须另一段代码也在操作该信号量
61      raw_spin_lock_irqsave(&sem->lock, flags);
62      if (likely(list_empty(&sem->wait_list)))
63          sem->count++;//如果信号量等待链表中为空，则对信号量值加1
64      else
65          __up(sem);//否则执行唤醒进程相关的操作
66      raw_spin_unlock_irqrestore(&sem->lock, flags);
    }
```

上述代码中的逻辑，已经描述了信号量的工作原理。需要注意的是，一个进程进入了 __down 函数中，设置了一个不可中断的等待状态，然后执行了 schedule_timeout 函数。这个执行了进程的调度器，就直接调度到别的进程运行了。

这时，这个进程就不会返回了，直到下一次它被 up 函数唤醒。执行了 wake_up_process 函数以后，重新调度它就会回到 schedule_timeout 函数下一行代码，沿着调用路经返回，最后从 __down 函数中出来，即进程睡醒了。

Linux 读写锁

在操作系统中，有很多共享数据，进程对这些共享数据要进行修改的情况很少，而读取的情况却是非常多的，这些共享数据的操作基本都是在读取。

如果每次读取这些共享数据都加锁的话，那就太浪费时间了，会降低进程的运行效率。因为读操作不会导致修改数据，所以在读取数据的时候不用加锁了，而是可以共享的访问，只有涉及到对共享数据修改的时候，才需要加锁互斥访问。

想像一下 100 个进程同时读取一个共享数据，而每个进程都要加锁解锁，剩下的进程只能等待，这会大大降低整个系统性能，这时候就需要使用一种新的锁了——读写锁。

读写锁也称为共享 - 独占（shared-exclusive）锁，当读写锁用读取模式加锁时，它是以共享模式上锁的，当以写入修改模式加锁时，它是以独占模式上锁的（互斥）。

读写锁非常适合**读取数据的频率远大于修改数据的频率**的场景中。这样可以在任何时刻，保证多个进程的读取操作并发地执行，给系统带来了更高的并发度。

那读写锁是怎么工作的呢？读写之间是互斥的，读取的时候不能写入，写入的时候不能读取，而且读取和写入操作在竞争锁的时候，写会优先得到锁，步骤如下。

1. 当共享数据没有锁的时候，读取的加锁操作和写入的加锁操作都可以满足。

2. 当共享数据有读锁的时候，所有的读取加锁操作都可以满足，写入的加锁操作不能满足，读写是互斥的。

3. 当共享数据有写锁的时候，所有的读取的加锁操作都不能满足，所有的写入的加锁操作也不能满足，读与写之间是互斥的，写与写之间也是互斥的。

如果你感觉刚才说的步骤还是太复杂，那我再给你画一个表，你就清楚了，如下所示。

锁的状态	读取的加锁操作	写入的加锁操作
无锁	允许	允许
持有读锁	允许	不允许
持有写锁	不允许	不允许

好了，我们明白了读写锁的加锁规则，现在就去看看 Linux 中的读写锁的实现，**Linux 中的读写锁本质上是自旋锁的变种。**

后面这段代码是 Linux 中读写锁的核心代码，请你注意，**实际操作的时候，我们不是直接使用上面的函数和数据结构，而是应该使用 Linux 提供的标准接口，如 read_lock、write_lock 等。**

复制代码

```
//读写锁初始化锁值
#define RW_LOCK_BIAS     0x01000000
//读写锁的底层数据结构
typedef struct{
    unsigned int lock;
}arch_rwlock_t;
//释放读锁
static inline void arch_read_unlock(arch_rwlock_t*rw){
    asm volatile(
        LOCK_PREFIX"incl %0" //原子对lock加1
        :"+m"(rw->lock)::"memory");
}
//释放写锁
```

```
static inline void arch_write_unlock(arch_rwlock_t*rw){
    asm volatile(
        LOCK_PREFIX"addl %1, %0"//原子对lock加上RW_LOCK_BIAS
        :"+m"(rw->lock):"i"(RW_LOCK_BIAS):"memory");
}
//获取写锁失败时调用
ENTRY(__write_lock_failed)
    //(%eax)表示由eax指向的内存空间是调用者传进来的
    2:LOCK_PREFIX addl  $ RW_LOCK_BIAS,(%eax)
    1:rep;nop//空指令
    cmpl $RW_LOCK_BIAS,(%eax)
    //不等于初始值则循环比较，相等则表示有进程释放了写锁
    jne   1b
    //执行加写锁
    LOCK_PREFIX subl  $ RW_LOCK_BIAS,(%eax)
    jnz 2b //不为0则继续测试，为0则表示加写锁成功
    ret //返回
ENDPROC(__write_lock_failed)
//获取读锁失败时调用
ENTRY(__read_lock_failed)
    //(%eax)表示由eax指向的内存空间是调用者传进来的
    2:LOCK_PREFIX incl(%eax)//原子加1
    1:  rep; nop//空指令
    cmpl  $1,(%eax) //和1比较 小于0则
    js 1b //为负则继续循环比较
    LOCK_PREFIX decl(%eax) //加读锁
    js  2b  //为负则继续加1并比较，否则返回
    ret //返回
ENDPROC(__read_lock_failed)
//获取读锁
static inline void arch_read_lock(arch_rwlock_t*rw){
    asm volatile(
        LOCK_PREFIX" subl $1,(%0)\n\t"//原子对lock减1
        "jns 1f\n"//不为小于0则跳转标号1处，表示获取读锁成功
        "call __read_lock_failed\n\t"//调用__read_lock_failed
        "1:\n"
        ::LOCK_PTR_REG(rw):"memory");
}
//获取写锁
static inline void arch_write_lock(arch_rwlock_t*rw){
    asm volatile(
        LOCK_PREFIX"subl %1,(%0)\n\t"//原子对lock减去RW_LOCK_BIAS
        "jz 1f\n"//为0则跳转标号1处
        "call __write_lock_failed\n\t"//调用__write_lock_failed
        "1:\n"
        ::LOCK_PTR_REG(rw),"i"(RW_LOCK_BIAS):"memory");
}
```

Linux 读写锁的原理本质是基于计数器，初始值为 0x01000000，获取读锁时对其减 1，结果不小于 0 则表示获取读锁成功，获取写锁时直接减去 0x01000000。

说到这里你可能要问了，为何要减去初始值呢？这是因为只有当锁值为初始值时，减去初始值结果才可以是 0，这是唯一没有进程持有任何锁的情况，这样才能保证获取写锁时是互斥的。

__read_lock_failed、__write_lock_failed 是两个汇编函数，注释写得很详细了，和前面自旋锁的套路是一样的。我们可以看出，读写锁其实是带计数的特殊自旋锁，能同时被多个读取数据的进程占有或一个修改数据的进程占有，但不能同时被读取数据的进程和修改数据的进程占有。

我们再次梳理一下获取、释放读写锁的流程，如下所示。

1. 获取读锁时，锁值变量 lock 计数减去 1，判断结果的符号位是否为 1。若结果符号位为 0 时，获取读锁成功，即表示 lock 大于 0。

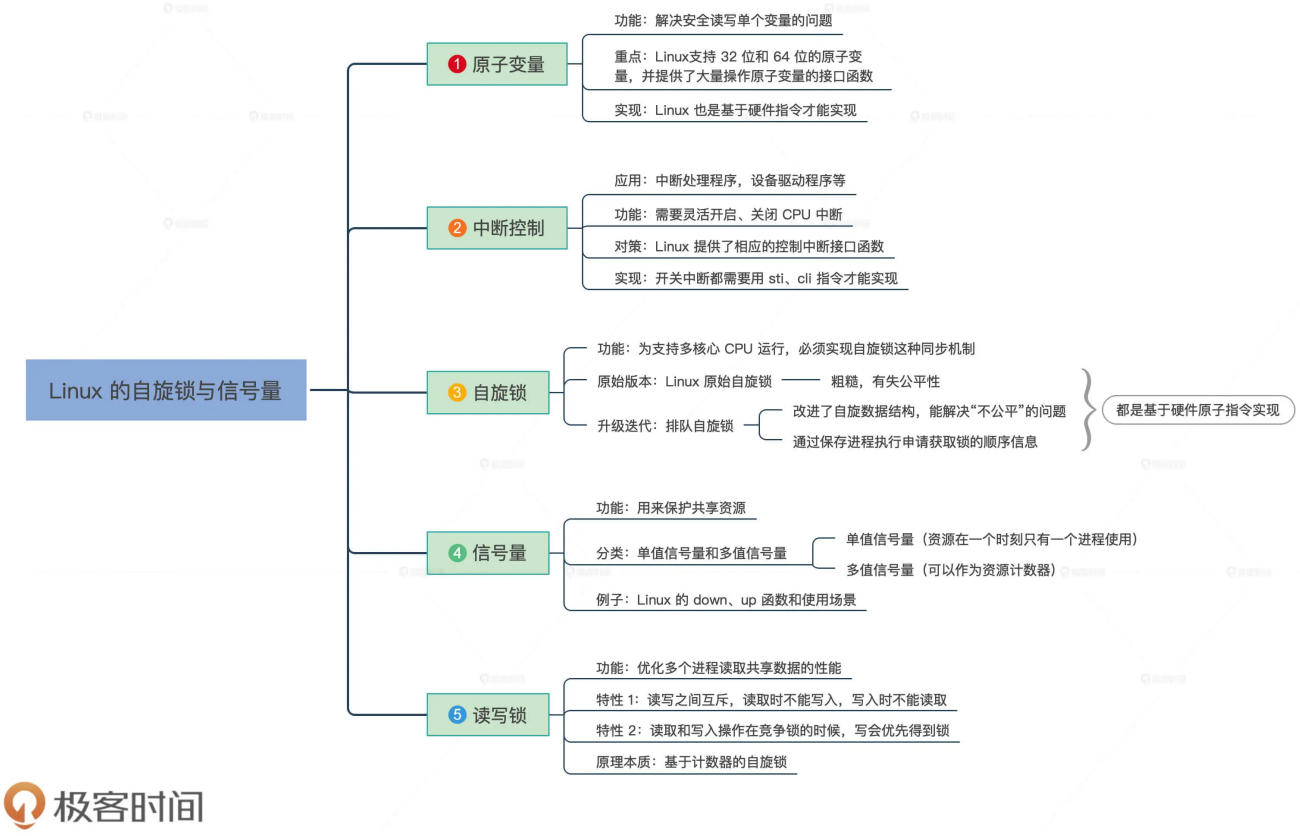
2. 获取读锁时，锁值变量 lock 计数减去 1，判断结果的符号位是否为 1。若结果符号位为 1 时，获取读锁失败，表示此时读写锁被修改数据的进程占有，此时调用 __read_lock_failed 失败处理函数，循环测试 lock+1 的值，直到结果的值大于等于 1。

3. 获取写锁时，锁值变量 lock 计数减去 RW_LOCK_BIAS_STR，即 lock-0x01000000，判断结果是否为 0。若结果为 0 时，表示获取写锁成功。

4. 获取写锁时，锁值变量 lock 计数减去 RW_LOCK_BIAS_STR，即 lock-0x01000000，判断结果是否为 0。若结果不为 0 时，获取写锁失败，表示此时有读取数据的进程占有读锁或有修改数据的进程占有写锁，此时调用 __write_lock_failed 失败处理函数，循环测试 lock+0x01000000，直到结果的值等于 0x01000000。

重点回顾

好了，这节课的内容讲完了。我们一起学习了 Linux 上实现数据同步的五大利器，分别是 Linux 原子变量、Linux 中断控制、Linux 自旋锁、Linux 信号量、Linux 读写锁。我把重点给你梳理一下。



锁，保证了数据的安全访问，但是它给程序的并行性能造成了巨大损害，所以在设计一个算法时应尽量避免使用锁。若无法避免，则应根据实际情况使用相应类型的锁，以降低锁的不当使用带来的性能损失。

思考题

请试着回答：上述 Linux 的读写锁，支持多少个进程并发读取共享数据？这样的读写锁有什么不足？

欢迎你在留言区和我交流，相信通过积极参与，你将更好地理解这节课的内容。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 8 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 锁：并发操作中，解决数据同步的四种方法

下一篇 10 | 设置工作模式与环境（上）：建立计算机

精选留言 (32)

写留言

老男孩 置顶
2021-05-28

这个排队自旋锁的实现方式感觉很风骚啊。关于读锁最大支持进程数是0x01000000（学友们都已经解答了）关于写饥饿的问题，既然写锁和读锁在同时获取锁状态时候写锁优先，那么就应该对读锁做一个限制，不能让读锁朝着最大数奔去。比如，系统检测到有写锁在等待，那么就限制新的读锁加入，等已经存在的读锁都释放了，写锁马上加锁更新资源。然后等待的读锁再开始加锁读取。这个等待的队列要分为读锁队列和写锁队列。优...

展开

作者回复: 嗯嗯见解独到

13

springXu 置顶
2021-05-30

同步与锁
操作系统是让应用程序认为当前有超大的内存空间和强大的cpu来工作的硬件环境，但其实硬件没有这么强大。那如何解决呢？比如在单核cpu上可以用分时技术，让进程交替执行。对于一个进程来说，我们可以把一个进程变成了多个线程来执行。但这样就产生了同一个资源可以是内存的某一具体地址，可以是鼠标可以是磁盘上的某一文件被多个线程...
展开

作者回复: 你好，铁子总结 到位

2 6

pedro
2021-05-28

以后我就是第一东吹了😁！
像这样的清晰明了，言简意赅的Linux内核源码解读实在是太少了，这样的文章读起来实在是太爽了，强烈小编安排一下东哥的下一个专栏叫做 《纵览Linux源码，小白也能学透》。
...
展开

作者回复: 哈哈 欢迎

19

blentle
2021-05-28

回答一下思考题
1.理论上可以支持x01000000这么多进程，但实际上受限于文件句炳也就是文件描述符的限制，还有考虑多个线程的问题等等，注定最终远远小于这个值
2.读写锁造成写饥饿的情况是不是可以参考jdk的读写锁的实现，在条件等待队列中判断队列第一个元素是不是一个写进程，如果是写进程，让其直接优先获取锁.
展开

作者回复: 嗯嗯

9

fhs
2021-05-30

读写锁部分中。假设t1时候1号写线程获取到了锁且不释放：lock=0，t2时刻2号写线程尝试获取：lock=-(0x0100 0000)，t3时刻1号读尝试获取：lock=-(0x0100 0001)。
之后1号写即便释放了锁，此时lock = -1。但是2号写的比较成功的条件是："lock+0x01000000，直到结果的值等于 0x01000000"。1号读的比较条件是"循环测试 lock+1 的值，直到结果的值大于等于 1"。...
展开

作者回复: 不是这样的哦 多看看课程

2

罗乾林
2021-05-28

Linux的读写锁，我的看法正好相反，写会造成读饥饿，一直有写入时读进程将会饥饿

作者回复: 你好，说说原因

1 1

吴国豪
2021-07-09

"2:\t" \ "cmpb $0,%0\n\t" \ //和0比较
"rep;nop\n\t" \ //空指令
"jle 2b\n\t" \ //小于或等于0跳转到标号2
这里最开始原子减1，如果小于0，也就是别的进程拿到锁的情况，那别的进程还锁后，应该是0吧。所以这里应该是判断是否小于0，而不是小于等于0？
展开

吴国豪
2021-07-09

"js 2f\n" \ //当结果小于0则跳转到标号2处，表示加锁失败
".section .text.lock,\"ax\"\n"

这里如果结果大于0，是不是执行完第二行就结束了，不会再进标号2了呢？

吴国豪
2021-07-07

Linux 排队自旋锁的部分，是怎么保证从读slock，判断，写slock的过程是原子的呢？我看好像分了很多行汇编。有没有可能两个进程同时读，然后判断成功，再写呢？

展开

作者回复: 不会的

Geek_4b6813

2021-07-06

最多支持同时有2^24个进程共享读锁(计算机常驻的进程基本上是不可能达到这个数的)存在的问题，写饥饿，只要有读进程存在，写进程就永远没机会获得锁。

作者回复: 是的

青玉白露

2021-07-01

从流程上来看，读锁最大为0x01000000；
由于读写锁互斥，如果有一个读，其他写都无法，只能自旋，浪费资源，可以考虑给写锁加队列，避免自旋。

展开

作者回复: 队列是有代价的

PawN

2021-06-27

有一点没太理解， 是在多核 的时候需要锁总线并且关中断吗， 单核只需要关中断就好了。 关中断时关当前进程的中断，还是所有进程的中断都关呢

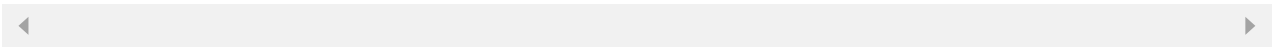
展开

作者回复: 單核下 同一時刻 只有一個進程运行

Feen
2021-06-17

到第9课的时候，发现留言的数量少了很多，感觉操作系统（或者计算机原理）的淘汰率真的很高，能出师的太少了。关于最后的问题，应该说读锁最大支持的进程数是0x01000000，写锁最大的进程数人为的设置为1。写锁也可以设置为0x01000000，因为对于计算机来说，到底有多少进程写对它来说无关，数据对计算机没有意义，数据只对于人能不能正确使用有关。所以才有上写锁的时候直接减去0x01000000这个初始值，目的是控制只有...
展开

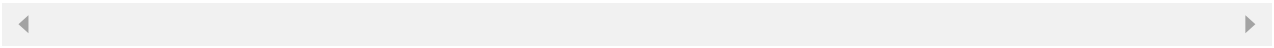
作者回复: 这只能表明你理解了计算机中的同步机制，但是操作系统需要知道的东西还有很多



K菌无惨
2021-06-13

请问“_cond_lock 只用代码静态检查工作”这句话时什么意思？

作者回复: 这是Linux那一套，它有一个检查工具，对代码进行静态检查，能及早发现问题



Geek_8c4220
2021-06-09

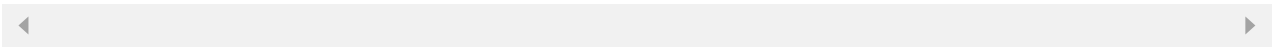
为什么这节里实现自旋锁的时候都没有关中断了呢？
展开

作者回复: 因为我没有讲

LT
2021-06-06

static inline int __raw_spin_trylock(raw_spinlock_t*lock){ int tmp; int new; asm volatile("movl %2,%0\n\t"//tmp=slock "movl %0,%1\n\t"//new=tmp "roll $16, %0\n\t"//tmp循环左移16位，即next和owner交换了 "cmpl %0,%1\n\t"//比较tmp和new即（owner、next）？=（next、owner） "jne 1f\n\t" //不等则跳转到标号1处 "addl $0x00010000, %1\n\t"//相当于next+1 "lock ; cmpxchgl %1,%2\n\t"//new和slock交换...
展开

作者回复: 这是旧版 linux的 新版本中没有了



Freddy
2021-06-01

Linux的读写锁支持0x01000000 = 16777216个进程并发读取共享数据；
Linux的读写锁本质上是自旋锁，进程没有成功获取到读锁或者写锁时，会进行自旋，浪费了CPU资源；

作者回复: 嘿嘿 是的呢

二三子也
2021-05-31

"lock ; xaddl %0, %1\n" //将inc和slock交换，然后 inc=inc+slock
老师，这个是不是应该是slock=slock+inc？
不然第一个申请的线程加不上锁啊？

作者回复: 是先交换

宏典
2021-05-31

重新定义一个代码段，避免后面的代码填充cache。因为大部分加锁都是成功的。
这个可以理解。
但是为何重新重新定义一个代码段，就可以保证后面的代码不填充cache?

作者回复: 是的是的

苏志辉
2021-05-31

"addl $0x00010000, %1\n\t"//相当于next+1

"lock ; cmpxchgl %1,%2\n\t"//new和slock交换比较
老师，这两句可以细讲一下吗？
是slock和谁比较，如果成功是把new给slock吗

作者回复: 嗯嗯